05 | 索引（下）：位移索引和时间戳索引的区别是什么?

2020-04-23 胡夕

讲述：胡夕
时长 14:16 大小 13.07M

你好，我是胡夕。今天，我们继续说索引那些事儿。

在上节课，我带你重点学习了 Kafka 源码中索引的抽象父类 AbstractIndex。我分析了 AbstractIndex 类的大体对象结构，还介绍了社区改进版的二分查找算法在 Kafka 索引上的应用。

前面说过，Kafka 索引类型有三大类：位移索引、时间戳索引和已中止事务索引。相比于最后一类索引，前两类索引的出镜率更高一些。在 Kafka 的数据路径下，你肯定看到过多.index 和.timeindex 后缀的文件。不知你是否有过这样的疑问：“这些文件是用来做什么的呢？” 现在我可以明确告诉你：.index 文件就是 Kafka 中的位移索引文件，而.timeindex 文件则是时间戳索引文件。

那么，位移索引和时间戳索引到底是做什么用的呢？它们之间的区别是什么？今天，我就为你揭晓这些问题的答案。

位移索引

在学习 Kafka 的任何一类索引的时候，我们都要关注两个问题：

1. 索引中的索引项是如何定义的？
2. 如何向索引写入新的索引项？

看到这里，你可能会很疑惑：“等等，难道我们不需要关心如何查询索引吗？” 当然需要啦！上节课我们不是讲过二分查找算法在索引中的应用了吗？如果你觉得有点生疏了，那就赶快先去复习一下吧。

现在，我们先来看下索引项的定义。

索引项的定义

位移索引也就是所谓的 OffsetIndex，它可是一个老资历的组件了。如果我没记错的话，国内大面积使用 Kafka 应该是在 0.8 时代。从那个时候开始，OffsetIndex 就已经存在了。每当 Consumer 需要从主题分区的某个位置开始读取消息时，Kafka 就会用到 OffsetIndex 直接定位物理文件位置，从而避免了因为从头读取消息而引入的昂贵的 I/O 操作。

在上节课，我提到过，不同索引类型保存不同的 <Key, Value> 对。就 OffsetIndex 而言，Key 就是消息的相对位移，Value 是保存该消息的日志段文件中该消息第一个字节的物理文件位置。

这里我来具体解释一下相对位移的含义。还记得 AbstractIndex 类中的抽象方法 entrySize 吗？它定义了单个 <Key, Value> 对所用的字节数。对于 OffsetIndex 来说，entrySize 就是 8，如 OffsetIndex.scala 中定义的那样：

复制代码

```
1     override def entrySize = 8
```

为什么是 8 呢？相对位移是一个整型（Integer），占用 4 个字节，物理文件位置也是一个整型，同样占用 4 个字节，因此总共是 8 个字节。

那相对位移是什么值呢？我们知道，Kafka 中的消息位移值是一个长整型（Long），应该占用 8 个字节才对。在保存 OffsetIndex 的 <Key, Value> 对时，Kafka 做了一些优化。每个 OffsetIndex 对象在创建时，都已经保存了对应日志段对象的起始位移，因此，OffsetIndex 索引项没必要保存完整的 8 字节位移值。相反地，它只需要保存与起始位移的差值（Delta）就够了，而这个差值是可以被整型容纳的。这种设计可以让 OffsetIndex 每个索引项都节省 4 个字节。

举个简单的例子。假设一个索引文件保存了 1000 个索引项，使用相对位移值就能节省大约 4MB 的空间，这是不是一件很划算的事情呢？

OffsetIndex 定义了专门的方法，用于将一个 Long 型的位移值转换成相对位移，如下所示：

复制代码

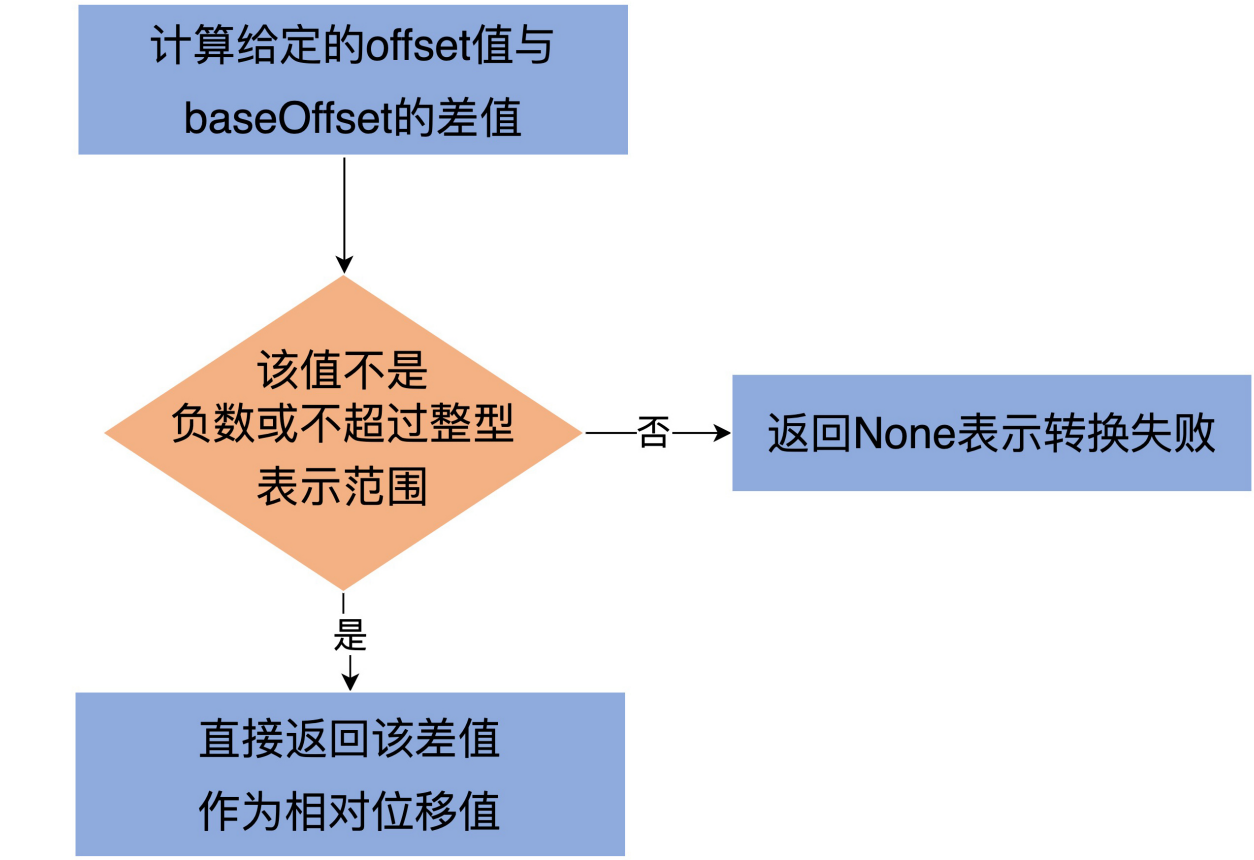
```
def relativeOffset(offset: Long): Int = {
    val relativeOffset = toRelative(offset)
    if (relativeOffset.isEmpty)
      // 如果无法转换成功（比如差值超过了整型表示范围），则抛出异常
      throw new IndexOffsetOverflowException(s"Integer overflow for offset: $o
    relativeOffset.get
}
```

relativeOffset 方法调用了父类的 toRelative 方法执行真正的转换。我们来看一下 toRelative 方法的实现。

复制代码

```
private def toRelative(offset: Long): Option[Int] = {
  val relativeOffset = offset - baseOffset
  if (relativeOffset < 0 || relativeOffset > Int.MaxValue)
    None
  else
    Some(relativeOffset.toInt)
}
```

逻辑很简单：第一步是计算给定的 offset 值与 baseOffset 的差值；第二步是校验该差值不能是负数或不能超过整型表示范围。如果校验通过，就直接返回该差值作为相对位移值，否则就返回 None 表示转换失败。



极客时间

现在，你知道 OffsetIndex 中的索引项为什么是 8 个字节以及位移值是如何被转换成相对位移了吧?

当读取 OffsetIndex 时，源码还需要将相对位移值还原成之前的完整位移。这个是在 parseEntry 方法中实现的。

复制代码

```
override protected def parseEntry(buffer: ByteBuffer, n: Int): OffsetPosition =
  OffsetPosition(baseOffset + relativeOffset(buffer, n), physical(buffer, n))
}
```

我来给你解释下具体的实现方法。

这个方法返回一个 OffsetPosition 类型。该类有两个方法，分别返回索引项的 Key 和 Value。

这里的 parseEntry 方法，就是要构造 OffsetPosition 所需的 Key 和 Value。Key 是索引项中的完整位移值，**代码使用 baseOffset + relativeOffset(buffer, n) 的方式将相对位移值还原成完整位移值**；Value 是这个位移值上消息在日志段文件中的物理位置，代码调用 physical 方法计算这个物理位置并把它作为 Value。

最后，parseEntry 方法把 Key 和 Value 封装到一个 OffsetPosition 实例中，然后将这个实例返回。

由于索引文件的总字节数就是索引项字节数乘以索引项数，因此，代码结合 entrySize 和 buffer.getInt 方法能够轻松地计算出第 n 个索引项所处的物理文件位置。这就是 physical 方法做的事情。

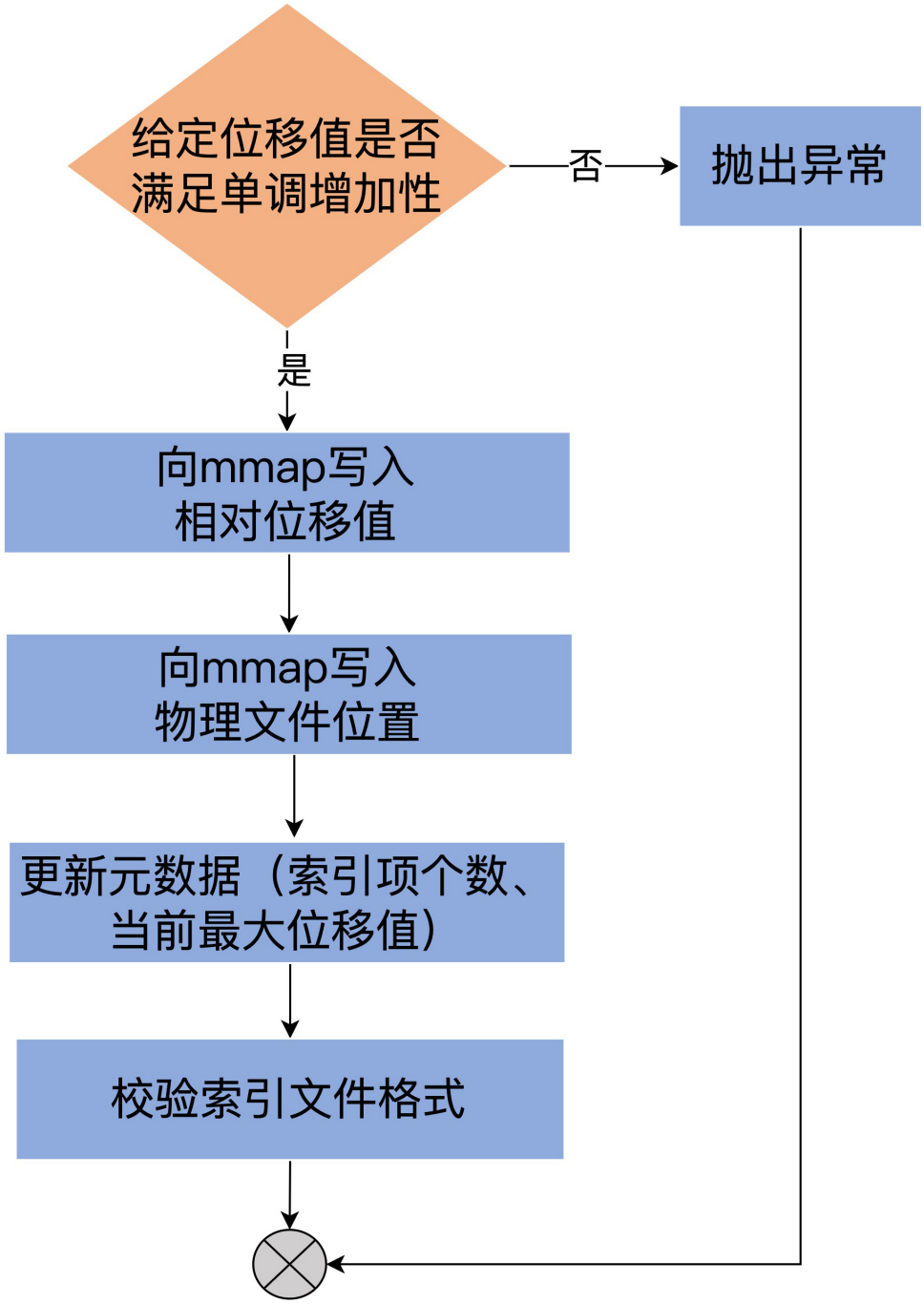
写入索引项

好了，有了这些基础，下面的内容就很容易理解了。我们来看下 OffsetIndex 中最重要的操作——**写入索引项 append 方法的实现。**

复制代码

```
def append(offset: Long, position: Int): Unit = {
  inLock(lock) {
    // 索引文件如果已经写满，直接抛出异常
    require(!isFull, "Attempt to append to a full index (size = " + _entries +
    // 要保证待写入的位移值offset比当前索引文件中所有现存的位移值都要大
    // 这主要是为了维护索引的单调增加性
    if (_entries == 0 || offset > _lastOffset) {
      trace(s"Adding index entry $offset => $position to ${file.getAbsolutePatl
      mmap.putInt(relativeOffset(offset)) // 向mmap写入相对位移值
      mmap.putInt(position) // 向mmap写入物理文件位置
      _entries += 1 // 更新索引项个数
      _lastOffset = offset // 更新当前索引文件最大位移值
      // 确保写入索引项格式符合要求
      require(_entries * entrySize == mmap.position(), s"$entries entries but
    } else {
      throw new InvalidOffsetException(s"Attempt to append an offset ($offset)
        s" the last offset appended (${_lastOffset}) to ${file.getAbsolutePath
    }
  }
}
```

append 方法接收两个参数：**Long 型的位移值**和 **Integer 型的物理文件位置。该方法最重要的两步，就是分别向 mmap 写入相对位移值和物理文件位置。**我使用一张图，来总结下 append 方法的执行流程：



除了 append 方法，索引还有一个常见的操作：截断操作（Truncation）。**截断操作是指，将索引文件内容直接裁剪掉一部分**。比如，OffsetIndex 索引文件中当前保存了 100 个索引项，我想只保留最开始的 40 个索引项。源码定义了 truncateToEntries 方法来实现这个需求：

复制代码

```
private def truncateToEntries(entries: Int): Unit = {
  inLock(lock) {
    _entries = entries
    mmap.position(_entries * entrySize)
    _lastOffset = lastEntry.offset
    debug(s"Truncated index ${file.getAbsolutePath} to $entries entries;" +
      s" position is now ${mmap.position()} and last offset is now ${_lastOffs
  }
}

```

这个方法接收 entries 参数，表示**要截取到哪个槽**，主要的逻辑实现是调用 mmap 的 position 方法。源码中的 _entries * entrySize 就是 mmap 要截取到的字节处。

下面，我来说说 OffsetIndex 的使用方式。

既然 OffsetIndex 被用来快速定位消息所在的物理文件位置，那么必然需要定义一个方法执行对应的查询逻辑。这个方法就是 lookup。

复制代码

```
def lookup(targetOffset: Long): OffsetPosition = {
  maybeLock(lock) {
    val idx = mmap.duplicate // 使用私有变量复制出整个索引映射区
    // largestLowerBoundSlotFor方法底层使用了改进版的二分查找算法寻找对应的槽
    val slot = largestLowerBoundSlotFor(idx, targetOffset, IndexSearchType.KEY
    // 如果没找到，返回一个空的位置，即物理文件位置从0开始，表示从头读日志文件
  // 否则返回slot槽对应的索引项
    if(slot == -1)
      OffsetPosition(baseOffset, 0)
    else
      parseEntry(idx, slot)
  }
}
```

我把主要的逻辑以注释的方式加到了代码中。该方法返回的，是不大于给定位移值 targetOffset 的最大位移值，以及对应的物理文件位置。你大致可以把这个方法，理解为位移值的 FLOOR 函数。

时间戳索引

说完了 OffsetIndex，我们来看另一大类索引：时间戳索引，即 TimeIndex。与 OffsetIndex 类似，我们重点关注 TimeIndex 中索引项的定义，以及如何写入 TimeIndex 索引项。

索引项的定义

与 OffsetIndex 不同的是，TimeIndex 保存的是 < 时间戳，相对位移值 > 对。时间戳需要一个长整型来保存，相对位移值使用 Integer 来保存。因此，TimeIndex 单个索引项需要占用 12 个字节。这也揭示了一个重要的事实：**在保存同等数量索引项的基础上，TimeIndex 会比 OffsetIndex 占用更多的磁盘空间**。

写入索引项

TimeIndex 也有 append 方法，只不过它叫作 maybeAppend。我们来看下它的实现逻辑。

复制代码

```
def maybeAppend(timestamp: Long, offset: Long, skipFullCheck: Boolean = false)
  inLock(lock) {
    if (!skipFullCheck)
      // 如果索引文件已写满，抛出异常
      require(!isFull, "Attempt to append to a full time index (size = " + _en
    // 确保索引单调增加性
    if (_entries != 0 && offset < lastEntry.offset)
      throw new InvalidOffsetException(s"Attempt to append an offset ($offset)
        s" the last offset appended (${lastEntry.offset}) to ${file.getAbsolut
    // 确保时间戳的单调增加性
    if (_entries != 0 && timestamp < lastEntry.timestamp)
      throw new IllegalStateException(s"Attempt to append a timestamp ($timest
        s" than the last timestamp appended (${lastEntry.timestamp}) to ${file

    if (timestamp > lastEntry.timestamp) {
      trace(s"Adding index entry $timestamp => $offset to ${file.getAbsolutePa
      mmap.putLong(timestamp) // 向mmap写入时间戳
      mmap.putInt(relativeOffset(offset)) // 向mmap写入相对位移值
      _entries += 1 // 更新索引项个数
```

```
20          _lastEntry = TimestampOffset(timestamp, offset) // 更新当前最新的索引项
21          require(_entries * entrySize == mmap.position(), s"${_entries} entries bu
22        }
23      }
24    }
```

和 OffsetIndex 类似，向 TimeIndex 写入索引项的主体逻辑，是向 mmap 分别写入时间戳和相对位移值。只不过，**除了校验位移值的单调增加性之外，TimeIndex 还会确保顺序写入的时间戳也是单调增加的。**

说到这里，我想到我当年读到这段代码时候的一个想法。那个时候，这段代码还没有加上时间戳单调增加的校验逻辑，我灵机一动，萌发了向 TimeIndex 写入一个过期时间戳的想法。一番动手操作之后，我真的向 TimeIndex 索引文件中写入了一个过期时间戳和位移。

你猜结果怎样？结果是引发了消费者端程序的彻底混乱。这是因为，当消费者端程序根据时间戳信息去过滤待读取的消息时，它读到了这个过期的时间戳并拿到了错误的位移值，因此返回了错误的数据。

为此，我还给社区提交了一个 Jira，当时被驳回了——理由是不允许向 TimeIndex 写入过期时间戳。跟你说这个趣事儿只是想说明，有的时候，读源码会诱发很多灵感或奇思妙想，而这些东西是你在平时使用过程中不会想到的。这也算是阅读源码的一大收获吧。

区别

讲到这里，这节课就接近尾声了。最后，我用一张表格汇总下 OffsetIndex 和 TimeIndex 的特点和区别，希望能够帮助你更好地理解和消化今天的重点内容。

区分项	OffsetIndex	TimeIndex
文件后缀名	.index	.timeindex
用途	根据位移值快速查找消息所在文件位置	根据时间戳快速查找特定消息的位移值
索引项	<相对位移值，物理文件位置>对	<时间戳，相对位移值>对
索引项字节数	8	12
Broker端参数	log.index.size.max.bytes	log.index.size.max.bytes

极客时间

总结

今天，我带你详细分析了 OffsetIndex 和 TimeIndex，以及它们的不同之处。虽然 OffsetIndex 和 TimeIndex 是不同类型的索引，但 Kafka 内部是把二者结合使用的。通常的流程是，先使用 TimeIndex 寻找满足时间戳要求的消息位移值，然后再利用 OffsetIndex 去定位该位移值所在的物理文件位置。因此，它们其实是合作的关系。

最后，我还想提醒你一点：**不要对索引文件做任何修改！**我碰到过因用户擅自重命名索引文件，从而导致 Broker 崩溃无法启动的场景。另外，虽然 Kafka 能够重建索引，但是随意地删除索引文件依然是一个很危险的操作。在生产环境中，我建议你尽量不要执行这样的操作。

课后讨论

OffsetIndex 中的 lookup 方法实现了类似于 FLOOR 函数的位移查找逻辑。你能否对应写一个类似于 CEILING 函数的位移查找逻辑，即返回不小于给定位移值 targetOffset 的最小位移值和对应的物理文件位置？

欢迎你在留言区畅所欲言，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　04 | 索引（上）：改进的二分查找算法在Kafka索引的应用

精选留言 (1)

写留言

胡夕 置顶

2020-04-23

你好，我是胡夕。我来公布上节课的“课后讨论”题答案啦～

上节课，咱们重点了解了Kafka索引对象以及改进版的二分查找算法，课后我让你思考下改进版二分查找算法中冷区和热区的分割线设定在8192字节处的原理。鉴于这是一个开放式的问题，我说下我的理解：就像源码注释里面写的那样，8192这个数字不大不小正合适...

展开

1